Backend Development Plan: Biz Hire MVP

This document outlines the architecture and API specification for the Biz Hire application backend, built with Node.js, Express, MySQL, and Sequelize.

1. Project Setup & Architecture

We will adopt a modular, layered architecture to ensure separation of concerns and maintainability.

A) Core Dependencies:

- **express**: Web framework for Node.js.
- **sequelize**: ORM for MySQL database interaction.
- **mysql2**: MySQL driver for Node.js.
- **jsonwebtoken**: For generating and verifying JWTs for authentication.
- **bcryptjs**: For hashing user passwords.
- **cors**: To handle Cross-Origin Resource Sharing.
- **dotenv**: To manage environment variables.
- **multer**: For handling file uploads (CVs/Resumes).
- **joi** or **express-validator**: For robust request validation.

B) Folder Structure:

codeCode

/biz-hire-backend

├── /config # Database config, environment variables

├── /controllers # Request/response logic for each route

├── /middleware # Auth, role checks, validation, error handling

├── /migrations # Sequelize database schema migrations

├── /models # Sequelize data models and associations

├── /routes # API route definitions (Express Router)

├── /seeders # Database seed files for initial data

├── /services # Business logic (e.g., PaymentService)

├── /uploads # Directory for storing uploaded CVs (can be replaced by cloud storage)

├── /utils # Helper functions (e.g., JWT helpers)

├── .env # Environment variables file

├── app.js # Main Express application setup

└── server.js # Server entry point

2. Database Models (Sequelize)

Based on the SRS Section 4, the Sequelize models will be defined as follows. These files will be generated using sequelize-cli.

- **models/user.js**:
    - email: STRING, unique, not null
    - passwordHash: STRING, not null
    - role: ENUM('Admin', 'Employer', 'Job Seeker'), not null
- **models/employerprofile.js**:
    - userId: INTEGER, foreign key to Users
    - companyName: STRING, not null
    - industry: STRING, not null
    - website: STRING, not null
    - logoUrl: STRING
    - description: TEXT
- **models/jobseekerprofile.js**:
    - userId: INTEGER, foreign key to Users
    - fullName: STRING, not null
    - cvUrl: STRING (path to the uploaded file)
    - structuredProfileData: JSON
- **models/job.js**:
    - employerProfileId: INTEGER, foreign key to EmployerProfiles
    - title: STRING, not null
    - description: TEXT, not null
    - location: STRING, not null
    - employmentType: ENUM('Full-time', 'Part-time', 'Contract', 'Internship'), not null
    - status: ENUM('Active', 'Filled', 'Closed'), default 'Active'
- **models/application.js**:

- jobId: INTEGER, foreign key to Jobs
- jobSeekerProfileId: INTEGER, foreign key to JobSeekerProfiles
- status: ENUM('New', 'Shortlisted', 'Interviewing', 'Hired', 'Rejected'), default 'New'

- **models/transaction.js**:
  - employerProfileId: INTEGER, foreign key to EmployerProfiles
  - jobId: INTEGER, foreign key to Jobs
  - amount: DECIMAL
  - paymentGatewayRefId: STRING
  - status: ENUM('Pending', 'Completed', 'Failed'), default 'Pending'

Associations will be defined in models/index.js or within each model file to establish the relationships described in SRS 4.2.

3. API Endpoint Design (RESTful Routes)

[auth] = Authentication middleware (valid JWT required)
[isEmployer] = Role middleware (user must be 'Employer')
[isJobSeeker] = Role middleware (user must be 'Job Seeker')
[isAdmin] = Role middleware (user must be 'Admin')

A) Authentication Routes (/api/auth)

- **POST /register**
  - **Description:** New user registration (FR-S-001).
  - **Body:** { "name": "...", "email": "...", "password": "...", "role": "Employer" }
  - **Response:** 201 Created with { "token": "...", "user": {...} }
- **POST /login**
  - **Description:** User authentication (FR-S-002).
  - **Body:** { "email": "...", "password": "..." }
  - **Response:** 200 OK with { "token": "...", "user": {...} }
- **POST /forgot-password** & **POST /reset-password**
  - **Description:** Password reset functionality (FR-S-003).

B) Profile Routes (/api/profiles)

- **POST /employer**
    - **Middleware:** [auth, isEmployer]
    - **Description:** Create an employer profile after registration (FR-E-001).
    - **Body:** { "companyName": "...", "industry": "...", ... }
    - **Response:** 201 Created with profile data.
- **POST /seeker**
    - **Middleware:** [auth, isJobSeeker]
    - **Description:** Create a job seeker profile, handles CV upload (FR-JS-001).
      Uses multipart/form-data.
    - **Body:** Form data including a cv file and optional structured data.
    - **Response:** 201 Created with profile data.
- **GET /me**
    - **Middleware:** [auth]
    - **Description:** Get the profile data for the currently logged-in user.
    - **Response:** 200 OK with the user's corresponding profile (Employer or Job Seeker).
- **PUT /employer** & **PUT /seeker**
    - **Middleware:** [auth] and respective role middleware.
    - **Description:** Update the profile of the currently logged-in user.
    - **Response:** 200 OK with updated profile data.

C) Job Routes (/api/jobs)

- **POST /**
    - **Middleware:** [auth, isEmployer]
    - **Description:** Create a new job posting (FR-E-002).
    - **Body:** { "title": "...", "description": "...", ... }
    - **Response:** 201 Created with job data.
- **GET /**
    - **Description:** Public endpoint to search and filter all active jobs (FR-JS-002). Supports query params like /api/jobs?location=Remote&q=Sales.

    - **Response:** 200 OK with { "jobs": [...], "totalPages": ..., "currentPage": ... }.

- **GET /my-jobs**
    - **Middleware:** [auth, isEmployer]
    - **Description:** Get a list of all jobs posted by the currently logged-in employer.
    - **Response:** 200 OK with job list.
- **GET /:jobId**
    - **Description:** Get details for a single job posting.
    - **Response:** 200 OK with job data.
- **PUT /:jobId**
    - **Middleware:** [auth, isEmployer]
    - **Description:** Update a job posting. The controller must verify the user owns the job.
    - **Response:** 200 OK with updated job data.

D) Application Routes (/api/applications)

- **POST /job/:jobId**
    - **Middleware:** [auth, isJobSeeker]
    - **Description:** Apply for a specific job (FR-JS-003).
    - **Response:** 201 Created with { "message": "Application submitted successfully." }.
- **GET /job/:jobId**
    - **Middleware:** [auth, isEmployer]
    - **Description:** Get all applicants for a specific job (ATS Dashboard) (FR-E-003). The controller must verify the user owns the job.
    - **Response:** 200 OK with a list of applicant profiles.
- **PUT /:applicationId/status**
    - **Middleware:** [auth, isEmployer]
    - **Description:** Update the status of an application (FR-E-004). This endpoint contains the core success fee logic.
    - **Body:** { "status": "Hired" }
    - **Response:** 200 OK. If status is "Hired", response includes a paymentUrl for the success fee transaction (FR-E-005).

E) Admin Routes (/api/admin)

- **GET /users**, **PUT /users/:userId**, **DELETE /users/:userId**
    - **Middleware:** [auth, isAdmin]
    - **Description:** Manage all user accounts (FR-A-002).
- **GET /jobs**, **PUT /jobs/:jobId**, **DELETE /jobs/:jobId**
    - **Middleware:** [auth, isAdmin]
    - **Description:** Manage all job postings on the platform (FR-A-003).
- **GET /transactions**
    - **Middleware:** [auth, isAdmin]
    - **Description:** View all success fee transactions (FR-A-004).

---

4. Core Logic & Implementation Details

- **RBAC (Role-Based Access Control):** The middleware [auth] will verify the JWT and attach the decoded user payload (including ID and role) to the req object. Subsequent role middleware (isEmployer, etc.) will check req.user.role to authorize access.
- **Success Fee Workflow (FR-E-005):**
    1. An Employer sends a PUT request to /api/applications/:applicationId/status with { "status": "Hired" }.
    2. The controller verifies the user is an Employer and owns the job associated with the application.
    3. The application's status is updated in the database.
    4. The associated job's status is updated to "Filled".
    5. A new Transaction is created with a Pending status.
    6. The controller calls a PaymentService module. This service interacts with the chosen payment gateway's API (e.g., Stripe) to create a payment session/link.
    7. The payment gateway's reference ID is saved to the Transaction.
    8. The payment link is returned in the API response to the frontend.
    9. A separate webhook endpoint (e.g., /api/webhooks/payment-status) will be created to listen for asynchronous updates from the payment gateway to mark the transaction as Completed or Failed. This is crucial for reliability.

- **File Uploads:** multer will be configured to handle multipart/form-data requests for CV uploads. It will save the file to a designated folder (or stream it to a cloud service like AWS S3) and attach the file path to the req object for the controller to save in the JobSeekerProfile model.